TERMINAL-UNIVERSE: TURNING AGENT TRAJECTORIES INTO SCALABLE TERMINAL ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

As terminal-based code agents become increasingly prevalent, the corresponding agent trajectories have accumulated at scale, while realistic, executable environments remain scarce. However, environments are what agent post-training actually requires: each one can be re-queried into many verifiable tasks and provides execution feedback, whereas a trajectory is a single frozen demonstration. Rather than generating environments from scratch, we observe that the tool-execution history in the existing agent trajectories exposes the structure and contents of the environments in which they ran, making it possible to reconstruct those environments from the trajectories themselves. Thus, we introduce **Terminal-Universe**, a framework that turns each trajectory into a reusable environment and explores it for synthesizing new tasks and continued interactions. Specifically, Terminal-Universe replays the file operations recorded in a trajectory to restore each file before the agent modified it, yielding a partial workspace; a completion agent then supplies the missing files and dependencies. On this recovered workspace, we both reconstruct the original intent task and synthesize entirely new ones. Besides, we also scale the tasks for the corresponding environment along two complementary axes: breadth and depth, to reproduce a routine pattern of real engineering practice. For breadth, we mine directional dependency relations between related environments and synthesize cross-workspace queries spanning multiple codebases, as developers routinely do in real-world development. For depth, we extend the initial single-turn query into a multi-round session that captures iterative user feedback and requirement refinement via a user agent. Applied to publicly available terminal agent trajectories, Terminal-Universe produces 37.3k task-sufficient environments. Supervised fine-tuning of Qwen3.5-27B on this corpus improves single-round performance on Terminal-Bench 2.1 by 11.9 points and multi-round performance on EvoCode-Bench v2 MT@4 by 13.8 points.

1 INTRODUCTION

As terminal-based code agents become increasingly prevalent, the trajectories they produce have accumulated at scale, while realistic, executable environments remain scarce. This gap matters because a trajectory and an environment are not equally useful. A trajectory is a single, fixed record: its quality is bounded by the responses of the policy model that produced it, and we cannot check whether the changes it made to the codebase are actually correct. An environment has none of these limits, because the same task can be solved again by a stronger model, the result can be verified by our own tests, and harder tasks can be posed on the same workspace. For post-training, the environment is therefore the resource worth scaling. Expert-authored benchmarks show what a good environment looks like. Terminal-Bench (Merrill et al., 2026), for example, pairs every task with a custom container, an instruction, and an executable verifier. This reliability comes from manual effort, which also limits how many tasks can be built, while effective post-training needs far more environments than manual curation can provide. Supplying realistic, verifiable environments at training scale is therefore the challenge we address.

Existing work scales executable environments in three ways. **Repository-based methods** build tasks from the git history of a real repository (Pan et al., 2024; Jain et al., 2025; Chen et al., 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

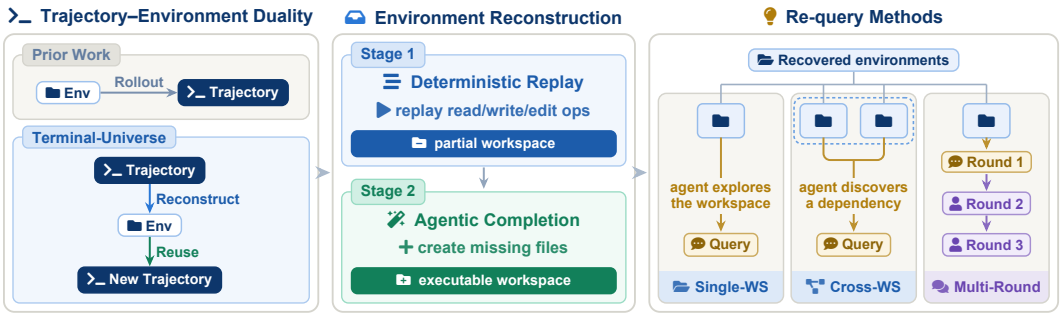


Figure 1: **Overview of Terminal-Universe.** A trajectory and an environment are two views of the same episode, so prior work rolls out a trajectory from an environment while we invert the mapping and recover an environment from a trajectory (left). Reconstruction proceeds in two stages, deterministic replay followed by agentic completion (middle). Each recovered environment is then re-queried within one workspace, across multiple dependent workspaces, or over multiple user rounds (right). Figure 2 details each mechanism.

For a bug that was fixed in the past, they use git to roll the repository back to the state just before the fix to form the environment, take the original bug report as the task, and reuse the tests that came with the fix to check the solution. **Perturbation methods** take a repository that passes its tests, inject a bug so that some tests now fail, and ask the agent to fix it (Yang et al., 2025; Lin et al., 2026). A few repositories can yield many tasks this way, but every task is a repair task, and the range of tasks is limited by the source repositories and the kinds of bugs that can be injected. **Task-conditioned synthesis** generates the task and its environment together from scratch, guided by categories or compositional axes (Gandhi et al., 2026; Ivison et al., 2026), skill taxonomies (Hua et al., 2026), or skill graph (Fan et al., 2026).

Across these routes, environment construction either starts from an existing environment or is coupled to a newly generated task from scratch. Trajectories, as observations of the environments they ran in, remain largely underexplored as a resource. The tool calls recorded inside a trajectory already reveal the content and structure of the environment it ran in. For instance, the tool `Read` shows file contents, and `Write` and `Edit` show how the workspace changed. This is enough to rebuild an executable copy of the original workspace. Therefore, we propose **Terminal-Universe** (Figure 1), a framework that turns each trajectory into a reusable environment and explores it for synthesizing new tasks and continued interactions.

Specifically, to construct the environments based on the trajectories (§ 3.1), we first replay the recorded file operations and restore each file to its state before the agent changed it, holding back the agent's own edits so that the workspace starts unsolved. Since replay usually yields only a partial workspace, a completion agent then automatically supplies the missing files and dependencies the task needs, without leaking the solution. For the corresponding tasks, we both reconstruct the original intent queries and synthesize entirely new ones. Besides, to better match real-world software engineering scenarios, we also scale the tasks on the environment along two complementary axes: breadth and depth (§ 3.2). **Breadth expansion** goes beyond the single-repository setting that most task synthesis assumes. We identify dependency relations between related environments and build cross-workspace tasks that span multiple codebases, which are much harder, e.g., reading a reference implementation, moving a feature from one project to another, or connecting two components. **Depth expansion** turns a single-round task into a multi-round one. After the agent completes an initial query, a user agent asks grounded follow-ups as the workspace changes—extending it with new requirements, or, when a round fails, asking for a fix based on what went wrong. We verify every round, and the outcome is fed back to the agent as natural, user-visible feedback. Every new task comes with a verifier written by an agent inside the container, and we keep only the trajectories whose tests all pass (§ 4).

On publicly available terminal agent trajectories, this pipeline produces 37.3k task-sufficient environments. Over these environments, we generate new queries and use Qwen3.7-Max as the teacher to roll out a solution trajectory for each task. Training Qwen3.5-27B on the resulting data validates the effectiveness of our method: it improves the single-round benchmark Terminal-Bench 2.1 (Terminal-Bench Team, 2026) by 11.9 points and the multi-round benchmark EvoCode-Bench v2 MT@4 by

13.8 points (§ 5). Extensive ablations confirm that each component contributes, most notably that re-solving tasks in reconstructed environments far outperforms imitating the raw trajectories.

To sum up, we make the following contributions:

- **Environments reconstruction.** We reframe recorded agent trajectories as a source of reusable executable environments, and reconstruct each one through deterministic replay followed by agentic completion—without needing the original repository or building from scratch.
- **Re-querying methods.** We scale the utility of each reconstructed environment through new task generation. Besides the vanilla single-workspace and single-turn tasks, we contribute breadth expansion via cross-workspace task synthesis and depth expansion via multi-round user queries. Each task is paired with an agent-authored verifier, and only trajectories that pass are kept.
- **Empirical validation.** Fine-tuning Qwen3.5-27B on the resulting corpus improves Terminal-Bench 2.1 by 11.9 points and EvoCode-Bench v2 MT@4 by 13.8 points over the same base model. Ablations confirm that each component contributes, and notably that re-solving in reconstructed environments far outperforms imitating the raw trajectories.

2 Related Work

Environment scaling for terminal agents. Existing methods scale executable workspaces through three main routes: recovering repository states from real development histories, modifying functioning workspaces, or constructing task-specific environments from generated specifications. SWE-Gym (Pan et al., 2024), R2E-Gym (Jain et al., 2025), and SWE-Universe (Chen et al., 2026) recover repository states tied to historical issues, commits, or pull requests. SWE-smith (Yang et al., 2025) and CLI-Gym (Lin et al., 2026) inject bugs or failures into working repositories or CLI workspaces. SETA (Shen et al., 2026b) scales verifiable terminal environments for reinforcement learning through synthesis and adaptive environment evolution. A separate family jointly generates terminal tasks and their containerized execution environments; we discuss their task-construction strategies below. Terminal-Universe instead takes recorded agent trajectories as its starting point. It replays the file operations recorded in each trajectory and uses agentic completion to restore missing project context, producing executable workspaces that can be reused. Table 23 in § F compares Terminal-Universe with representative methods.

Terminal task synthesis. Existing methods derive tasks either from abstract priors or from concrete executable environments. Top-down pipelines generate task-environment pairs from categories or seeds (Gandhi et al., 2026; Pi et al., 2026; Zhu et al., 2026), and structured variants organize generation around capability taxonomies, compositional axes, or skill graphs (Hua et al., 2026; Ivison et al., 2026; Fan et al., 2026). Others use agent skills, executable meta-tasks, or solver feedback to diversify and calibrate tasks (Cheng et al., 2026; Pan et al., 2026; Meng et al., 2026).

Environment-grounded methods instead derive tasks from working code and project context: they perturb healthy environments, mine repository documentation, ground tasks in real issues, or jointly build instructions, solutions, and verifiers in a shared environment (Lin et al., 2026; Wu et al., 2026a; Yang et al., 2026b; Shi et al., 2026). RST recursively expands verified seeds by lengthening solutions and realigning their tasks, verifiers, and environments (Li et al., 2026b). T1 (Yang et al., 2026a) trains a long-horizon terminal agent with reinforcement learning on tasks synthesized this way. Terminal-Universe instead reconstructs the workspaces of past trajectories and generates new tasks within and across them.

Interactive and multi-round agents. Recent benchmarks move from single-turn prompts to multi-turn interaction. InterCode formalizes interactive coding with execution feedback in containerized shells (Yang et al., 2023). SWE-INTERACT (Raghavendra et al., 2026) uses a simulated user to reveal requirements step by step and request revisions, and SWE-Together (Wu et al., 2026b) replays real user–agent sessions through a state-conditional user simulator. ICAE-Bench (Peng et al., 2026b) evaluates interactive project construction from incomplete requirements with an automated user agent, and EvoCode-Bench v2 (Shen et al., 2026a) evaluates agents on sequential development requests in persistent workspaces. Terminal-Universe brings this setting to training data through depth expansion, which extends reconstructed workspaces into multi-round sessions that capture iterative user feedback and requirement refinement.

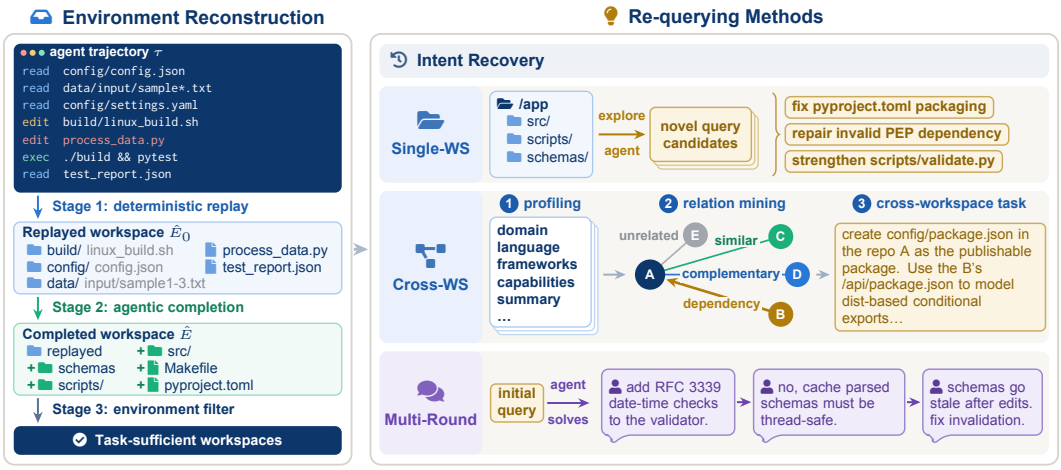


Figure 2: **Framework of Terminal-Universe.** Left: a trajectory is replayed into a partial workspace, completed into an executable one, and kept only if it is sufficient for its task. Right: the four re-querying mechanisms recover the source task, or synthesize new tasks within one workspace, across related workspaces, and over multiple user rounds.

3 Terminal-Universe

Figure 2 details the mechanisms of environment reconstruction and re-querying. The following subsections describe environment reconstruction (§ 3.1), re-querying (§ 3.2), and verification (§ 3.3).

3.1 Environment Reconstruction

From the file and command operations recorded in a trajectory τ, we recover an executable workspace $\widehat{E}$ that approximates the environment E in which τ was produced. Recovery is lossy, because unaccessed files, implicit system dependencies, and network resources leave no direct trace. We therefore reconstruct in three stages.

Stage 1: deterministic replay. Replay processes the read, write, and edit operations in τ in order and records, for each accessed path, the earliest and latest contents visible in the trajectory. The initial workspace $\widehat{E}_0$ keeps each pre-existing file at its earliest version, before the agent's first change. Files created by the agent are excluded, and the agent's changes are kept separately for later verification. Since τ exposes only the paths the agent touched, and may show a file or terminal output only in part, $\widehat{E}_0$ remains partial.

Stage 2: agentic completion. Given $\widehat{E}_0$ and the recovered task q, a completion agent creates missing files, completes partial ones, and restores dependencies so that q becomes solvable, without implementing it (§ C.2). We apply this stage to all replayed workspaces and denote the result by $\widehat{E}$; § 4 reports its effect on workspace complexity.

Stage 3: environment filtering. A completed workspace is useful only when it exposes enough project context to support its task. An agentic judge inspects each $\widehat{E}$ with read-only shell and file tools and labels it sufficient or insufficient for q, based on whether its source, configuration, data, and structure give a capable agent enough context (§ C.3). Only sufficient workspaces are kept (§ 4).

Each workspace runs in a standard `ubuntu:24.04` container with network access. Compared with repository-specific images, this lowers cost and simplifies deployment, with a modest drop in resolve rate reported by prior work (Zeng et al., 2026).

3.2 Re-querying

Reconstruction yields executable environments, and re-querying decides how well they are used. We use four mechanisms: Intent Recovery reconstructs the source task, Single-WS synthesizes new tasks

within one workspace, Cross-WS adds breadth by connecting related workspaces, and Multi-Round adds depth by extending a query into an interactive session.

Intent recovery. Consolidate one or more source user requests into a self-contained task.

We normalize each source trajectory into a chronological stream of user requests, agent actions, and file changes. In a single-round trajectory, the sole substantive request defines the task. In a multi-round trajectory, the first substantive request sets the topic, and later requests are merged when they clarify, constrain, or extend it; unrelated task shifts are dropped (§ D.1).

Single-workspace synthesis. Derive new queries from individual reconstructed workspaces.

An offline agent inspects each workspace and proposes five self-contained candidate tasks under groundedness, structural diversity, and verifiability constraints. We randomly pick one valid candidate per environment (§§ D.2 and E.1).

Breadth expansion. Connect related workspaces in a single query.

Cross-workspace synthesis. We first discover directional dependency relations across recovered environments. An agent profiles each workspace's technical domain and implemented capabilities. We retrieve candidate pairs by TF-IDF nearest-neighbor search, and an LLM judge keeps directional edges where a target workspace lacks a capability that a reference workspace already implements. Each pair yields exactly one task, which combines a writable target workspace with a read-only reference mounted at a separate path. The agent confirms that the gap is real and specifies observable target behaviors that close it, checkable with deterministic local commands. The query gives only these behaviors and the reference's mount path, not its internals, so the solver must find, understand, and adapt the reference implementation on its own (§§ D.3 and E.2).

Depth expansion. Extend solved terminal tasks through user-agent follow-ups.

Multi-round user queries. Starting from an initial terminal query, we keep the workspace across rounds and add a user agent after the first response. Two mechanisms keep the sessions coherent and realistic:

Evolving task specification. The user agent maintains an explicit requirement tracker logging active, satisfied, and updated requirements. Prior to each follow-up, it updates this tracker by adding, modifying, or replacing constraints, maintaining contextual coherence as the workspace evolves.

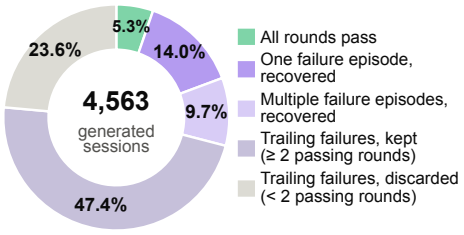


Figure 3: Pass/fail patterns in Multi-Round.

Round-level verification and feedback. At each extension round, the user agent commits the updated specification, prompting an automated verifier to author round-level acceptance tests before the coding agent acts. Upon receiving the agent's response, the test runner evaluates both new criteria and active regression checks. The solver agent is strictly isolated from test scripts and tracebacks; the user agent interprets the structured test results and translates any failure into natural, user-observable complaints. Intermediate failures are retained within the conversation history, providing realistic supervision for error diagnosis and recovery.

Follow-up requests take three styles: feature extension, which adds grounded requirements after a passing round; feature revision, which asks for a fix after a failure; and feature conflict, which changes earlier requirements to reflect new user intent. The style follows from the round outcome and the session context, and we report the observed distribution (Table 14; §§ D.4 and E.3). Sessions run for up to six follow-up rounds or until a termination sentinel. After the selection in § 3.3, the 3,079 retained sessions average 4.51 rounds, and 69.6% contain a failure that is later repaired (Figure 3).

3.3 Verification and Filtering

Agentic verifier construction. Each Single-WS or Cross-WS task gets an executable verifier written by a dedicated agent inside the target container. Given the task specification and the workspace files,

the verifier agent writes a self-contained pytest suite and refines it through iterative local runs. The tests check only the public interfaces and expected behaviors specified in the task prompt (§ D.5).

Solution rollout. The teacher model[1] solves each task with the Claude Code scaffold (Anthropic, 2026) inside the task container. It decodes at temperature 1.0 with interleaved thinking over a 256k-token context, for up to 500 agent turns within four hours; the other settings match our evaluation (§ A). The container has network access to fetch missing dependencies.

Verification and data selection. For Single-WS and Cross-WS, the test suite runs on the final workspace, and a trajectory is kept only if all tests pass. Multi-Round is selected per round: we trim trailing failed rounds from terminated sessions and keep a session only if it has at least two verified passing rounds. Failures that precede a successful recovery are kept as supervision. Accepted trajectories are formatted as multi-turn SFT demonstrations and decontaminated against the evaluation benchmarks (§ A).

4 Data Construction at Scale

Seed selection. We source raw agent trajectories from terminal-style CLI and software-engineering corpora and keep a trace as a seed if the workspace observed at its end has at least 5 files and 100 lines. To prevent leakage, we remove all Terminal-Bench-derived sources. Complete source breakdowns and selection statistics are detailed in § C.1.

Reconstruction statistics. The pipeline yields 68,263 reconstructed environments. Replayed terminal workspaces start with few files, since solution files from the original rollout are withheld. Agentic completion raises their mean file count from 2.9 to 22.4, and also expands the richer SWE workspaces (Figure 7 and Table 18 in § C.2). After contamination filtering and repository-level deduplication, we apply the Stage 3 judge to 38,294 terminal environments and 1,900 SWE repositories (one reconstruction each).

Table 1: Workspace sufficiency under recovered intents.

Pool	# Environments	Sufficient post-replay (%)	Sufficient post-completion (%)
Terminal	38,294	40.2	**93.5**
SWE	1,900	20.1	**77.1**

Replay alone makes 40.2% of terminal and 20.1% of SWE workspaces sufficient, and completion raises these rates to 93.5% and 77.1% (Table 1), leaving 37,273 sufficient environments for task generation. The terminal pool is mostly Python (84.7%), followed by C++ and C, and data processing, DevOps, and security make up over 80% of its domains (Figure 6 in § C.1).

SFT corpus composition. On these environments, Intent Recovery reconstructs the original tasks, Single-WS yields novel within-repository tasks, Cross-WS discovers inter-repository dependency challenges, and Multi-Round extends initial queries into iterative sessions. After verifier filtering, the terminal pool yields 31,977 SFT demonstrations (25,386 Single-WS, 3,512 Cross-WS, and 3,079 Multi-Round trajectories), totaling about 1.42B training tokens. Median interaction statistics are provided in § B.4.

5 Experiments

We fine-tune Qwen3.5-27B by SFT to obtain Terminal-Universe-27B (§ A). Table 2 reports single- and multi-round results. Trained on the Full Mixture (the verifier-filtered Single-WS, Cross-WS, and Multi-Round trajectories, 32.0k records), it reaches 52.8% on Terminal-Bench 2.0 and 58.1% on Terminal-Bench 2.1 under Terminus2-XML (+11.2 and +11.9 over the base), and 58.2% on Terminal-Bench 2.1 under Claude Code (+10.4). On EvoCode-Bench v2, it raises MT@4 from 6.3 to 20.1 and Case score from 67.8 to 76.1, so the gains carry over to persistent tasks with cumulative requirements.

[1]All model-driven components in this work use Qwen3.7-Max (Qwen Team, 2026) (xhigh effort) as the teacher model. We also report results with GLM-5.3 (GLM-5-Team et al., 2026) as the teacher in § B.3.

Table 2: Terminal-Bench and EvoCode-Bench v2 results (%). Terminal-Bench columns report Avg. Pass@1. **Green bold**: best among task synthesis methods; blue underline: second best. *: measured by us on the released models (§ A); other scores are from the original reports.

Model	Base Model	Data size	TB Scaffold	Terminal-Bench		EvoCode-Bench v2	
				TB2.0	TB2.1	MT@4	Case Score
Base and teacher models							
Qwen3.5-27B	–	–	Terminus2	41.6	46.2	6.3*	67.8*
Qwen3.7-Max	–	–	Terminus2	69.7	74.5	39.8*	83.4*
Terminal task synthesis methods							
TerminalTraj-32B (Wu et al., 2026a)	Qwen2.5-Coder-32B	50.7k	Terminus2	22.0	28.5*	0.0*	15.9*
TermiGen-32B (Zhu et al., 2026)	Qwen2.5-Coder-32B	3.3k	BashAgent	19.3	21.3*	0.0*	12.2*
LiberCoder-235B (Lin et al., 2026)	Qwen3-235B-A22B	48.3k	OpenHands	31.0	–	–	–
Nemotron-Terminal-32B (Pi et al., 2026)	Qwen3-32B	490k	Terminus2	27.4	27.9*	0.0*	13.7*
SkillSynth-32B (Fan et al., 2026)	Qwen3-32B	10.7k	Terminus2	29.6	–	–	–
Terminal-World-32B (Cheng et al., 2026)	Qwen3-32B	5.7k	Terminus2	31.5	–	–	–
Terminal-Lego-32B (Yang et al., 2026b)	Qwen3-32B	15.3k	Terminus2	24.3	–	–	–
CLI-Universe-32B (Hua et al., 2026)	Qwen3-32B	6k	Terminus2	33.4	–	–	–
TMax-27B (Ivison et al., 2026)	Qwen3.6-27B	14.6k	Vanillux2Agent	42.7	44.9	17.1*	72.5*
OpenThinker-Agent-32B (Raoof et al., 2026)	Qwen3-32B	100k	Terminus2	26.2	30.7*	0.0*	19.7*
Meta-Task-32B (Pan et al., 2026)	Qwen3-32B	3.2k	Terminus2	31.8	–	–	–
RST-27B (Li et al., 2026b)	Qwen3.5-27B	37.5k	Terminus2	49.4	–	–	–
CalibForge-35B-A3B (Meng et al., 2026)	Qwen3.5-35B-A3B	5.4k	CalibForge-Eval	47.6	–	–	–
FACET-Terminal-27B (Shi et al., 2026)	Qwen3.5-27B	1.2k	Terminus2	–	47.6	–	–
Terminal-Universe-27B	Qwen3.5-27B	32.0k	Terminus2	**52.8**	**58.1**	**20.1**	**76.1**

6 Analysis and Ablation Studies

This section isolates the contribution of each design choice. We ask whether re-solving beats training on the original trajectories (§ 6.1), whether agentic completion is needed beyond replay (§ 6.2), whether verifier filtering is worth the data it discards (§ 6.3), and how to spend a fixed budget across breadth (§ 6.4), depth (§ 6.5), and more environments versus more queries (§ 6.6). We also test transfer beyond terminal workspaces (§ B.1).

6.1 Task Re-solving

This ablation asks whether reconstruction is needed at all, that is, whether re-solving a recovered task beats imitating the original trajectory (Table 3). Intent Recovery is not verifier-filtered here, so verifier selection plays no role.

Table 3: Terminal-Bench 2.1 comparison of source-trajectory SFT and Intent Recovery (%).

Variant	Data size	Claude Code	Terminus2-XML	Avg.
Qwen3.5-27B	-	$47.8_{2.0}$	$46.2_{3.9}$	47.0
Source trajectories	35.8k	$33.0_{3.9}$	$40.3_{2.1}$	36.7
Intent Recovery	35.8k	$\mathbf{51.3}_{3.1}$	$\mathbf{52.9}_{1.4}$	**52.1**

Re-solving improves over source-trajectory SFT. Both variants use the same Qwen3.5 chat template and tool-call schema. Source SFT keeps the original agents' behavior, while Intent Recovery uses a stronger teacher to re-solve recovered tasks in reconstructed workspaces. Source SFT averages 36.7 and Intent Recovery 52.1, suggesting that demonstrations regenerated under one consistent teacher give more effective supervision.

6.2 Impact of Agentic Completion

This ablation asks whether agentic completion earns its cost. We train on Intent Recovery corpora whose environments come from replay only: the replay-only variant uses the 35,809 source instances that replay reconstructs, and the completed variant is the production Intent Recovery corpus at the same volume. Both share the recovered queries and teacher configuration.

Agentic completion helps. Training on completed environments beats replay-only environments by 4.2 points (52.9 versus 48.7; Table 4) and varies less across runs (±1.4 versus ±3.5). This matches Table 1: replay alone leaves most terminal workspaces insufficient, and completion restores the context that re-solving needs.

Table 4: Effect of agentic completion.

Variant	Records	Terminus2-XML
Qwen3.5-27B	–	$46.2_{3.9}$
Replay only	35.8k	$48.7_{3.5}$
Replay + agentic completion	35.8k	$\mathbf{52.9}_{1.4}$

Replay-only supervision still improves over the base model. The replay-only variant gains 2.5 points over the base model (46.2 to 48.7). Given an incomplete workspace, the teacher often repairs missing files or setup first, so part of its supervision targets workspace repair rather than the recovered task.

6.3 Role of Verifier Filtering

This ablation asks whether discarding verifier-failed trajectories is worth the data it removes. For each synthesis method, we compare training on all teacher trajectories against training only on the verifier-passed subset from the same launched-task pool.

Table 5: Effect of verifier filtering.

Variant	Records	Terminus2-XML
Single-WS w/o verifier	35.1k	$56.0_{3.3}$
Single-WS w/ verifier	25.4k	$\mathbf{56.4}_{2.6}$
Cross-WS w/o verifier	7.1k	$53.2_{2.1}$
Cross-WS w/ verifier	3.5k	$\mathbf{55.4}_{2.7}$

Verifier filtering matters more for harder tasks. On Single-WS data, the verifier-passed subset performs similarly to the full corpus (56.4 versus 56.0) with fewer records. On Cross-WS data, retaining verifier-failed trajectories lowers performance to 53.2, whereas the verifier-passed subset reaches 55.4 with less than half the data. The effect is also visible in Multi-Round synthesis: removing round-level verifier supervision lowers both EvoCode-Bench metrics (Table 7).

6.4 Breadth Expansion

This ablation asks whether cross-workspace tasks add supervision beyond single-workspace synthesis (Table 6). Cross-WS alone reaches 55.4 under Terminus2-XML, and adding it to Single-WS raises the score from 56.4 to 58.4.

Table 6: Terminal-Bench 2.1 results for Single-WS, Cross-WS, and their mixture (%).

Variant	Data size	Terminus2-XML
Qwen3.5-27B	–	$46.2_{3.9}$
Single-WS	25.4k	$56.4_{2.6}$
Cross-WS	3.5k	$55.4_{2.7}$
Single-WS + Cross-WS	28.9k	$\mathbf{58.4}_{2.1}$

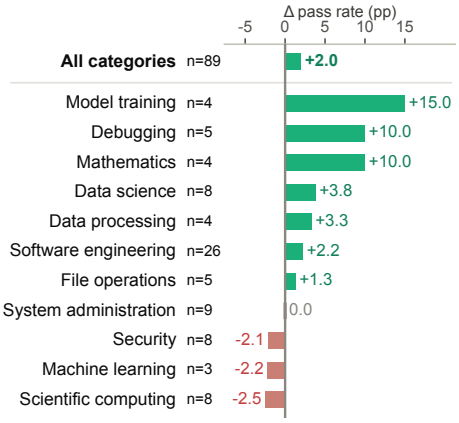


Figure 4: Per-category Terminal-Bench 2.1 pass-rate changes when Cross-WS data is added to Single-WS training under Terminus2-XML. Single-task categories are omitted.

How Cross-WS expands breadth. Single-WS synthesizes one task inside each recovered environment. Cross-WS instead requires changes in a writable workspace based on evidence from a related read-only workspace. This yields domain–operation combinations absent from single-workspace tasks and requires the agent to combine information across codebases (Table 16 in § B.4).

Where cross-workspace data helps. Figure 4 breaks the gain from $56.4_{2.6}$ to $58.4_{2.1}$ down by task category. Software engineering, the largest category with 26 tasks, rises from 46.9 to 49.1, and larger gains appear in model training (+15.0) and debugging (+10.0). Three categories drop by 2.1 to 2.5

points, within run-to-run variation for categories of at most eight tasks. Together with the overall gain, this suggests that cross-workspace data adds useful supervision.

6.5 Multi-Round Depth Expansion

This ablation asks whether Multi-Round data helps on persistent, cumulative tasks, and whether round-level verification matters. We evaluate on EvoCode-Bench v2 (Table 7).

Multi-round data improves depth. Adding Multi-Round data to Single-WS raises MT@4 from 18.4 to 21.0 and Case score from 71.9 to 76.9: the model completes longer runs of requests before its first failure and passes more verifier cases across the session.

Round-level verification directs effective continuation. Removing round-level verifier feedback lowers MT@4 by 2.2 points and Case score by 3.7 points. Without a grounded signal of what is still wrong, continuation becomes less directed and yields longer but lower-quality trajectories.

Table 7: Effect of Multi-Round synthesis and round-level verification on EvoCode-Bench v2 (%).

Variant	Data size	MT@4 score	Case score
Single-WS	25.4k	18.4	$71.9_{3.5}$
Single-WS + Multi-Round	28.5k	**21.0**	$\mathbf{76.9}_{1.4}$
Single-WS + Multi-Round (w/o round verifier)	28.5k	18.8	$73.2_{2.6}$

6.6 Expansion-Axis Ablation

This ablation asks where a fixed data budget is best spent. From a common Single-WS pool of 17,558 environments with one query and one teacher solution each (17.6k records), we double the corpus along one axis: a second set of environments, a second query per environment, or a second solution per query. All variants reach about 35k records and use the setup of § A, so differences reflect allocation rather than size.

Table 8: Data-scaling ablation for Single-WS synthesis. The base pool is doubled with more environments, queries per environment, or solutions per query.

Variant	Envs	Queries / env	Solutions / query	Records	Terminus2-XML
Base pool	17,558	1	1	17.6k	$53.2_{4.1}$
Environment expansion	35,116	1	1	35.1k	$\mathbf{56.0}_{3.3}$
Query expansion	17,558	2	1	35.1k	$53.8_{3.3}$
Solution expansion	17,558	1	2	35.1k	$53.9_{2.4}$

Environment expansion gives the largest gain, from 53.2 to 56.0, while more queries or solutions leave the score nearly unchanged (53.8 and 53.9). Each new environment brings a distinct executable context, whereas a second query or solution on the same workspace mostly repeats what the model has seen. This supports our focus on reconstructing more environments.

7 Conclusion

We present Terminal-Universe, a framework that turns recorded agent trajectories from fixed demonstrations into reusable execution environments. Deterministic replay and agentic completion restore each workspace, on which we synthesize new tasks and expand them across workspaces for breadth and across rounds for depth. Re-solving recovered tasks outperforms SFT on the source trajectories, and verifier-selected Single-WS, Cross-WS, and Multi-Round data improve both single- and multi-round terminal-agent performance. Recorded trajectories can thus be repurposed into interactive environments, a scalable source of grounded agent training data that does not require building environments from scratch. We discuss limitations in § F.1.

AI USE STATEMENT

Large language models are part of the method studied in this paper. All model-driven components of the pipeline, including completion, sufficiency judging, task synthesis, verifier construction, the user agent, and solution rollout, use Qwen3.7-Max (§ 3). The authors used AI assistants to help write code for the experimental infrastructure and to edit the wording and layout of this paper. The authors designed the study, checked all results, and take full responsibility for the content.

REFERENCES

Anthropic. Claude Code overview. https://code.claude.com/docs/en/overview, 2026. Accessed 2026-07-14.

Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zhou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. CoderForge-Preview: SOTA open dataset for training efficient agents, February 2026. URL https://www.together.ai/blog/coderforge-preview. Project core leads: Alpay Ariyak, Zhongzhu Zhou, and Qingyang Wu.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, Binyuan Hui, and Zhongxin Liu. SWE-Universe: Scale real-world verifiable environments to millions, 2026. URL https://arxiv.org/abs/2602.02361.

Zihao Cheng, Hongru Wang, Zeming Liu, Xinyi Wang, Xiangrong Zhu, Yuhang Guo, Wei Lin, Jeff Z. Pan, and Yunhong Wang. Terminal-World: Scaling terminal-agent environments via agent skills, 2026. URL https://arxiv.org/abs/2605.20876.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiangtao Guan, Yun Yang, Dingxin Hu, Jiang Zhou, Xing Wu, Zhuo Han, Feng Zhang, and Lilin Wang. Toward scalable terminal task synthesis via skill graphs, 2026. URL https://arxiv.org/abs/2604.25727.

Kanishk Gandhi, Shivam Garg, Noah D. Goodman, and Dimitris Papailiopoulos. Endless terminals: Scaling RL environments for terminal agents, 2026. URL https://arxiv.org/abs/2601.16443.

GLM-5-Team, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui,

Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. GLM-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. SWE-Factory: Your automated factory for issue resolution training data and evaluation benchmarks, 2026. URL https://arxiv.org/abs/2506.10954.

gyung. LFM2-Terminal-SFT-Processed. Hugging Face dataset, 2026. URL https://huggingface.co/datasets/gyung/LFM2-Terminal-SFT-Processed. Accessed 2026-08-23.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-Universe: Towards verifiable task synthesis engine for terminal agents, 2026. URL https://arxiv.org/abs/2606.22883.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. Tmax: A simple recipe for terminal agents, 2026. URL https://arxiv.org/abs/2606.23321.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents, 2025. URL https://arxiv.org/abs/2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, Xin Zhang, Zijian Jin, Bowen Li, Chaoyun Zhang, Yu Kang, Yufan Huang, Elsie Nallipogu, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. RepoLaunch: Automating build and management of code repositories across languages and platforms, 2026a. URL https://arxiv.org/abs/2603.05026.

Zhongzhi Li, Yucheng Shi, Zongxia Li, Ruhan Wang, Anhao Li, Zixun Huang, Junyao Yang, Lei Ke, Ninghao Liu, Haitao Mi, and Leowei Liang. Recursive synthesis for long-horizon terminal tasks, 2026b. URL https://arxiv.org/abs/2608.05466.

Yusong Lin, Haiyang Wang, Shuzhe Wu, Lue Fan, Feiyang Pan, Sanyuan Zhao, and Dandan Tu. CLI-Gym: Scalable CLI task generation via agentic environment inversion, 2026. URL https://arxiv.org/abs/2602.10999.

Fanzhe Meng, Guoxin Chen, Jiale Zhao, Shuang Sun, Zhiyu Lin, Wayne Xin Zhao, Ruihua Song, Ji-Rong Wen, and Kai Jia. CalibForge: Adversarial solver calibration for scaling learnable terminal tasks, 2026. URL https://arxiv.org/abs/2608.06352.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu,

Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym, 2024. URL https://arxiv.org/abs/2412.21139.

Zhihong Pan, Jiyuan He, Kai Zhang, Yupeng Han, Ze Liu, Yuze Zhao, Yongcong Ye, and Zhaohua Yang. Meta-Task: Turning terminal task synthesis into a terminal task for scalable agent training, 2026. URL https://arxiv.org/abs/2607.27929.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. LiteCoder-Terminal: Scaling long-horizon terminal environments for learning language agents, 2026a. URL https://arxiv.org/abs/2605.29559.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, and Yixin Cao. ICAE-Bench: Evaluating coding agents as interactive project builders, 2026b. URL https://arxiv.org/abs/2607.21217.

Renjie Pi, Grace Lam, Mohammad Shoeybi, Pooya Jannaty, Bryan Catanzaro, and Wei Ping. On data engineering for scaling LLM terminal capabilities, 2026. URL https://arxiv.org/abs/2602.21193.

Qwen Team. Qwen3.7: The agent frontier, May 2026. URL https://qwen.ai/blog?id=qwen3.7.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-INTERACT: Reimagining SWE benchmarks as user-driven long-horizon coding sessions, 2026. URL https://arxiv.org/abs/2606.30573.

Negin Raoof, Richard Zhuang, Marianna Nezhurina, Etash Guha, Atula Tejaswi, et al. OpenThoughts-Agent: Data recipes for agentic models, 2026. URL https://arxiv.org/abs/2606.24855.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCode-Bench: Evaluating coding agents in multi-turn iterative interactions, 2026a. URL https://arxiv.org/abs/2605.24110.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, Zhichen Zeng, Jiajun Li, Shi Dong, Yueming Yuan, Boyuan Ma, Qizheng Zhang, Jiwei Fu, Yuzhen Mao, Wendong Fan, Ping Nie, Philip Torr, Bernard Ghanem, Changran Hu, Jonathan Lingjie Li, Urmish Thakker, and Guohao Li. SETA: Scaling environments for terminal agents, 2026b. URL https://arxiv.org/abs/2607.10891.

Kou Shi, Zun Wang, Qisheng Su, Shiting Huang, Ziao Zhang, Zhen Fang, Qingnan Ren, Jin Liu, Yu Zeng, Yiming Zhao, Lin Chen, Zehui Chen, and Feng Zhao. FACET: Preserving source intent and executable state in terminal task synthesis, 2026. URL https://arxiv.org/abs/2608.18580.

Terminal-Bench Team. Terminal-Bench 2.1. https://www.tbench.ai/news/terminal-bench-2-1, May 2026. Release announcement, accessed 2026-07-14.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-scale terminal agentic trajectory generation from dockerized environments, 2026a. URL https://arxiv.org/abs/2602.01244.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions, 2026b. URL https://arxiv.org/abs/2606.29957.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. InterCode: Standardizing and benchmarking interactive coding with execution feedback, 2023. URL https://arxiv.org/abs/2306.14898.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, 2025. URL https://arxiv.org/abs/2504.21798.

Junyao Yang, Yucheng Shi, Zhongzhi Li, Ruhan Wang, Zongxia Li, Haitao Mi, and Leowei Liang. T1: Terminal agent reinforcement learning for long-horizon tasks, 2026a. URL https://arxiv.org/abs/2609.11042.

Sidi Yang, Chaofan Tao, Jierun Chen, Tiezheng Yu, Ruoyu Wang, Yuxin Jiang, Yiming Du, Wendong Xu, Jing Xiong, Taiqiang Wu, Lifeng Shang, Xiaohui Li, Ngai Wong, and Haoli Bai. What makes interaction trajectories effective for training terminal agents?, 2026b. URL https://arxiv.org/abs/2606.03461.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents, 2026. URL https://arxiv.org/abs/2606.28436.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. TermiGen: High-fidelity environment and robust trajectory synthesis for terminal agents, 2026. URL https://arxiv.org/abs/2602.07274.

Appendix

A Experimental Setup 15

B Additional Experiments and Analyses 16

B.1 Cross-domain Generalization 16

B.2 Transfer to Software Engineering Benchmarks 16

B.4 SFT Corpus Statistics 17

C Data Sources and Environment Reconstruction 18

C.1 Data Sources and Selection 18

C.2 Reconstruction and Completion 19

C.3 Workspace Sufficiency Evaluation 21

D Re-querying and Verification Details 22

D.1 Intent Recovery 23

D.2 Single-WS Synthesis 23

D.3 Cross-WS Synthesis 24

D.4 Multi-Round User Queries 28

D.5 Verifier Construction 29

E Case Studies 30

E.1 Reconstruction and Single-Workspace Re-querying 31

E.2 Cross-WS Synthesis 33

E.3 Multi-Round User Queries 34

F Position of This Paper, Discussion and Limitations 36

F.1 Discussion and Limitations 36

A EXPERIMENTAL SETUP

Training. We fine-tune Qwen3.5-27B by SFT on Terminal-Universe data for two epochs. We use a constant learning rate of 7×10^{-6}, a global batch size of 256, and a 256k-token sequence length. Before training, we run a 13-gram contamination check against Terminal-Bench tasks and exclude Terminal-Bench-derived source datasets from all four re-querying variants.

Single-round evaluation (Terminal-Bench 2.0 and 2.1). We evaluate Terminal-Bench with Claude Code (version 2.1.126) and Terminus2 (Merrill et al., 2026) using the XML parser. Evaluation mirrors the decoding and execution configurations of the solution rollout (§ 3.3), employing temperature 1.0, top-$p = 0.95$, interleaved thinking, a 256k context window, a 65,536-token turn limit, proactive summarization at 176k tokens, up to 500 agent turns, and a four-hour wall-clock timeout. For Claude Code, interactive and web-retrieval tools (`WebFetch`, `WebSearch`, `AskUserQuestion`, `EnterPlanMode`, `ExitPlanMode`) are disabled. Each container runs with 12 CPU cores and 32 GiB of memory. Reported scores represent mean pass rates over six independent runs. Differences between Terminal-Bench 2.0 and 2.1 are documented in the benchmark update (Terminal-Bench Team, 2026).[2]

Multi-round evaluation (EvoCode-Bench v2). We evaluate depth expansion on EvoCode-Bench v2[3] (Shen et al., 2026a), which contains 26 coding tasks and 227 rounds (5–15 requests per task). Workspaces and sessions persist, and cumulative verifiers check both current and active prior requirements. Our Qwen3.5-27B checkpoints use Terminus2-XML at temperature 1.0 with interleaved thinking, a 256k-token context, a per-turn limit of 65,536 tokens, proactive summarization at 176k tokens, up to 500 agent turns per request round, and a total wall-clock limit of 10 hours per stateful task. Each task is evaluated in four independent runs. MT@4 uses fail-stop scoring: a run receives credit only for the consecutive rounds it passes before its first failure, and each task-round is credited if any run reaches it successfully. The score averages credited rounds within each task and then across tasks. Case score averages verifier-case pass rates within tasks and then across tasks and runs to measure partial progress.

Table 9 lists the configurations used for Terminal-Universe-27B and the five reproduced open-weight baselines in Table 2. These reproduced baselines use model-specific settings rather than a shared evaluation configuration. The listed settings apply to both Terminal-Bench 2.1 and EvoCode-Bench v2 unless noted otherwise. For TMax-27B, the listed configuration applies only to our EvoCode-Bench v2 reproduction; its Terminal-Bench scores and Vanillux2Agent scaffold follow the original report. Qwen3.5-27B and Qwen3.7-Max use the Terminal-Universe-27B configuration, with xhigh effort enabled for Qwen3.7-Max. On EvoCode-Bench v2, TermiGen-32B instead uses Terminus2-JSON with an 81,920-token summarization threshold.

Table 9: Evaluation configurations used in our reproductions.

Setting	Nemotron-Terminal-32B	OpenThinker-Agent-32B	TermiGen-32B	TerminalTraj-32B	TMax-27B	Terminal-Universe-27B
Architecture	`Qwen3ForCausalLM`	`Qwen3ForCausalLM`	`Qwen2ForCausalLM`	`Qwen2ForCausalLM`	`Qwen3_5ForConditionalGeneration`	`Qwen3_5ForConditionalGeneration`
Scaffold	Terminus2	Terminus2	BashAgent	Terminus2	Terminus2	Terminus2
Parser	JSON	JSON	Text JSON	JSON	XML	XML
Context length	40,960	32,768	131,072	32,768	262,144	262,144
Turn max tokens	8,192	8,192	8,192	8,192	65,536	65,536
Summ. threshold	32,960	28,672	disabled	24,768	176k	176k
Temperature	0.6	0.6	0.6	0.7	0.7	1.0
Top-p / top-k	0.95 / 20	0.95 / 20	0.8 / 20	0.8 / 20	0.95 / 20	0.95 / 20
Interleaved thinking	✗	✗	✗	✗	✓	✓
Thinking enabled	✓	✓	✗	✗	✓	✓
Shared across all reproductions						
TB wall clock	4 h					
EvoCode wall clock	10 h					
TB runs	6					
EvoCode runs	4					

[2] https://github.com/harbor-framework/terminal-bench-2/pull/53

[3] https://unipat.ai/benchmarks/EvoCode-Bench

B Additional Experiments and Analyses

B.1 Cross-domain Generalization

This ablation asks whether the pipeline helps beyond the terminal domain it was built on. We apply Intent Recovery to trajectories from software-engineering (SWE) workspaces rather than terminal ones, then fine-tune and evaluate under the setup of § A, measuring transfer on Terminal-Bench 2.1. We use Intent Recovery alone, as the simplest re-querying variant, to keep the test clean.

Table 10: Terminal-Bench 2.1 pass rates (%) after training on SWE intent-recovery trajectories.

Variant	Data size	Claude Code	Terminus2-XML	Avg.
Qwen3.5-27B	-	$47.8_{2.0}$	$46.2_{3.9}$	47.0
Intent Recovery	10.3k	$\mathbf{50.6}_{0.9}$	$\mathbf{49.4}_{1.6}$	**50.0**

Of the 1,900 SWE repositories we evaluate, 1,464 are judged task-sufficient. Because a repository may contribute multiple source tasks, these repositories yield approximately 10.3k Intent Recovery training trajectories. Training on them improves the Terminal-Bench average from 47.0 to 50.0 (Table 10), and both scaffolds move in the same direction (Claude Code 47.8 $\rightarrow$ 50.6, Terminus2-XML 46.2 $\rightarrow$ 49.4), so the gain is not tied to a single evaluation setup. This shows that trajectories synthesized from SWE workspaces provide supervision useful for terminal-agent behavior, not only for their own source distribution. We test the SWE-to-terminal direction here; § B.2 tests the reverse.

B.2 Transfer to Software Engineering Benchmarks

This experiment tests the reverse direction of § B.1: whether training on terminal data also helps repository-level issue resolution. We evaluate the Full Mixture and the Breadth Mixture (Single-WS + Cross-WS, § 6.4), both trained only on data from the terminal pool, on SWE-bench Pro (Deng et al., 2025) and SWE-bench Multilingual (Yang et al., 2025). SWE-bench Pro contains long-horizon tasks from real repositories, and we use its public set of 731 tasks. SWE-bench Multilingual contains 300 tasks from 42 repositories in nine languages other than Python. All models run under Claude Code (version 2.1.126) with temperature 1.0, top-p = 0.95, top-k = 20, interleaved thinking, a 256k context window, a 64,000-token turn limit, up to 300 agent turns, and a four-hour timeout per attempt. Each task is attempted three times, and we report the mean resolved rate over the three runs.

Table 11: Resolved rates (%) on SWE-bench Pro and SWE-bench Multilingual under Claude Code. Subscripts give the standard deviation across three runs.

Model	SWE-bench Pro	SWE-bench Multilingual
Qwen3.5-27B	$46.4_{0.8}$	$59.9_{1.3}$
Full Mixture	$49.6_{1.3}$	$\mathbf{64.8}_{2.8}$
Breadth Mixture	$\mathbf{50.9}_{1.4}$	$63.5_{2.4}$

Terminal data transfers to software engineering. Both mixtures improve over the base model on both benchmarks (Table 11). On SWE-bench Pro, the resolved rate rises from 46.4 to 49.6 with the Full Mixture and to 50.9 with the Breadth Mixture. On SWE-bench Multilingual, it rises from 59.9 to 64.8 and 63.5, respectively. The Multilingual gains are notable because Python accounts for 84.7% of the reconstructed terminal pool (Figure 6), while none of the Multilingual tasks use Python.

B.3 GLM-5.3 as the Teacher

This experiment tests whether the pipeline depends on one teacher model. We rerun Intent Recovery with GLM-5.3 (GLM-5-Team et al., 2026) as the teacher. The recovered task descriptions are reused from the main pipeline, and GLM-5.3 performs agentic completion, verifier construction, and solution rollout. Solution rollouts use high effort and the other roles use max effort. A trajectory is kept only if it passes its verifier, which leaves 8.4k records. We fine-tune Qwen3.5-27B on them for two epochs with a 128k-token sequence length. We evaluate on Terminal-Bench 2.1 with Terminus2-XML, and GLM-5.3 itself is evaluated with high effort. Scores for GLM-5.3 and the model trained on its trajectories are averaged over four runs.

Table 12: Terminal-Bench 2.1 pass rates (%) under Terminus2-XML with GLM-5.3 as the teacher.

Model	Data size	Terminus2-XML
Qwen3.5-27B	-	$46.2_{3.9}$
GLM-5.3 (teacher)	-	$86.8_{2.1}$
Intent Recovery (GLM-5.3)	8.4k	$53.9_{0.9}$

Training on the GLM-5.3 trajectories raises the pass rate from 46.2 to 53.9 (Table 12), so the pipeline also provides useful supervision with a teacher from a different model family.

B.4 SFT Corpus Statistics

Table 13 summarizes the final terminal SFT corpora, with turns, tool calls, and tokens reported as per-record medians. Multi-Round records are substantially longer because they extend a completed session with follow-up user requests. Table 14 gives the distribution of interaction styles over the continuation rounds of the retained Multi-Round records. We impose no target ratio over the styles. Table 15 breaks the Multi-Round records down by round. Follow-up rounds take about as many turns and tool calls as the initial round.

Table 13: Final SFT corpus statistics after length filtering.

Dataset	Records	Turns	Tool calls	Tokens
Intent Recovery	35,809	12	17	36.1k
Single-WS	25,386	14	20	30.4k
Cross-WS	3,512	23	38	46.5k
Multi-Round	3,079	92	102	126.1k

Table 14: Interaction styles across Multi-Round continuation rounds.

Interaction style	Share
Feature extension	62.7%
Feature revision	29.3%
Feature conflict	8.0%

Table 15: Per-round statistics of the Multi-Round records. Records count the records that reach each round; turns and tool calls are per-round medians.

Round	1	2	3	4	5	6	7
Records	3,079	3,060	2,673	2,251	1,636	791	254
Turns	19	22	19	20	20	19	19
Tool calls	25	25	20	21	21	20	20

Table 16 contrasts the two synthesis variants on the same teacher. Cross-workspace tasks yield substantially longer and more complex trajectories: 1.6× the assistant turns, 1.9× the tool calls, and 1.5× the tokens per record at the median. They are also harder for the teacher: pass@1 drops from 72.3% on single-workspace tasks to 49.2%. Grounding the requirement in a second codebase forces the agent to read and reconcile two repositories before editing, lengthening trajectories and raising task difficulty rather than merely adding data volume.

Table 16: Trajectory and difficulty profiles of Single-WS and Cross-WS synthesis.

Variant	Turns		Tool calls		Tokens / record		Teacher pass@1
	Median	Mean	Median	Mean	Median	Mean	
Single-WS	14	15.1	20	21.2	30.4k	32.9k	72.3%
Cross-WS	23	25.3	38	39.7	46.5k	51.9k	49.2%

C Data Sources and Environment Reconstruction

C.1 Data Sources and Selection

Figure 5 summarizes the core reconstruction and sufficiency stages.

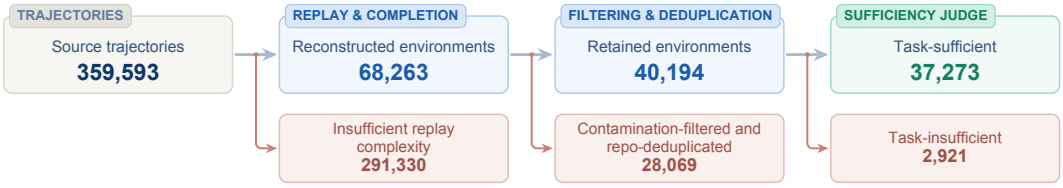


Figure 5: Core environment-reconstruction flow.

Table 17 summarizes each included source before and after reconstruction. Whether a trajectory can be reconstructed depends on the file evidence it exposes: multi-file editing traces reveal enough project context to rebuild, whereas command-heavy CLI traces often expose few file contents. We exclude Terminal-Bench-derived corpora and sources dominated by view-only operations or unsupported action formats.

SWE deduplication and decontamination. Public SWE corpora often contain multiple rollouts or formats of the same task. We deduplicate within each source by repository, base commit, and problem statement, selecting the trajectory whose replay exposes the most workspace content. We then remove instances from SWE-bench Verified repositories (Jimenez et al., 2024) and records that we cannot link to a source trajectory. Table 17 gives the resulting per-source counts. Figure 6 shows the language and domain composition of the reconstructed terminal workspaces.

Table 17: Source corpora and reconstructed environments. The Trajectories column counts source trajectories drawn from each corpus.

Source corpus	Pool	License	Trajectories	Environments
SWE-rebench (Badertdinov et al., 2025)	SWE	CC-BY-4.0	67,074	6,118
SWE-smith (Yang et al., 2025)	SWE	MIT	95,851	8,476
CoderForge (Ariyak et al., 2026)	SWE	Apache-2.0	32,964	3,978
SWE-Gym (Pan et al., 2024)	SWE	MIT	4,152	929
LFM2-Terminal (gyung, 2026; Pi et al., 2026)	Terminal	CC-BY-4.0	139,841	46,037
LiteCoder-Terminal (Peng et al., 2026a)	Terminal	MIT	19,711	2,725
Total			**359,593**	**68,263**

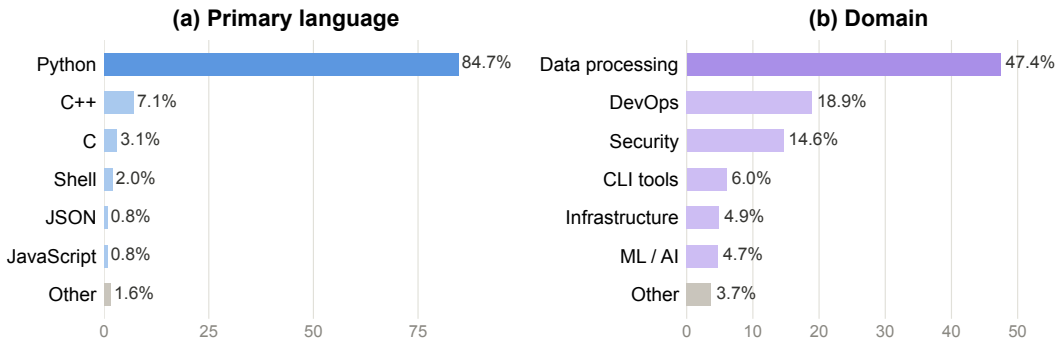


Figure 6: Language and domain composition of reconstructed terminal workspaces.

C.2 Reconstruction and Completion

The completion agent receives the recovered request, partial workspace, and file inventory. It fills missing project context by creating files, completing partial files, and configuring dependencies needed by the existing project. It must not implement the requested task or reveal where its solution should be made.

Packaging. Reconstructed projects are placed under `/app`. Files accessed elsewhere in the filesystem are stored separately.

PROMPT Environment Completion

ROLE & GOAL

```
You are a workspace completion agent.  Complete a Docker workspace so that the
given task becomes solvable, but NOT solved.
```

RUNTIME INPUTS

```
Task the future agent must solve:
<ORIGINAL_INSTRUCTION>

Current workspace files (container-absolute paths; project root and working
directory:  /app):
<DOCKER_LAYOUT_FILES>
```

PROMPT Environment Completion *(continued)*

COMPLETION CONTRACT

```
Objectives
1. Make the task solvable. Create missing project files or complete partial
ones, including configs, fixtures, scripts, data, schemas, manifests, and
support code that the task clearly depends on.
2. Do NOT solve the task. Do not create the requested implementation, fix,
feature, configuration change, or output.
3. Create realistic project context with consistent naming, plausible
structure, and enough working surrounding code for a future agent to
investigate.

Rules
- Prefer source code and data, followed by build/config files and referenced
scripts. Create documentation only when explicitly required or consumed at
runtime.
- Existing non-target code may be substantial, but do not add tests for the
not-yet-implemented target or comments that reveal its exact solution
location.
- Keep code syntactically valid and consistent with the workspace's language,
framework, versions, and style.
- Use network access or install packages only when needed to restore
dependencies required by the existing project.
- You may create missing files and complete partial files, but do not make the
change requested by the task.
```

Completion quality. Manual inspection of 30 randomly sampled Terminal completions found that 22 contained only task-relevant support files, while eight introduced substantial files or code not needed to support the task. We found no task solutions in the 30 completed workspaces.

Figure 7 and Table 18 compare workspace size before and after agentic completion.

Replayed size tracks source exploration. The size of a replayed workspace reflects the source trajectory rather than a limitation of replay. Replay restores only files that the source agent observed or edited, so its output grows with how extensively the agent explored. SWE agents explore their repositories more, and their replayed workspaces already average 5.6 files and 487 code lines, versus 2.9 files and 43 code lines for terminal workspaces (Table 18).

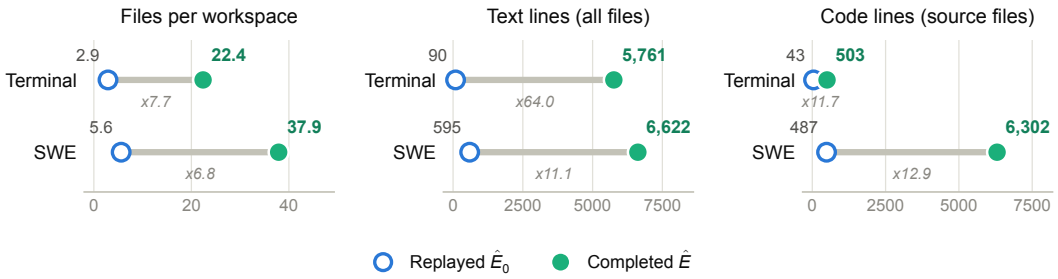


Figure 7: Mean workspace size before and after agentic completion.

Table 18: Workspace complexity before and after agentic completion. Cells report median / mean.

	Terminal pool		SWE pool	
	Replayed ($\widehat{E}_0$)	**Completed** ($\widehat{E}$)	**Replayed** ($\widehat{E}_0$)	**Completed** ($\widehat{E}$)
Files per workspace	2 / 2.9	13 / 22.4	5 / 5.6	21 / 37.9
Text lines (all files)	60 / 90	539 / 5,761	537 / 595	1,854 / 6,622
Code lines (source files)	0 / 43	316 / 503	421 / 487	1,643 / 6,302

C.3 Workspace Sufficiency Evaluation

Evaluation scope. We evaluate every terminal reconstruction that survives decontamination and repository-level deduplication. For SWE, we evaluate one representative reconstruction and its Intent Recovery task per repository. The agentic judge inspects each workspace with read-only shell and file tools.

Scoring. A workspace is task-sufficient when its source, configuration, data, and structure give a capable agent enough context to work on the task. Dependencies, caches, generated outputs, and optional documentation that can be recreated are not required. The judge evaluates available project context rather than build success.

Table 19 gives the corresponding source-level breakdowns.

Table 19: Source breakdown of workspace sufficiency under recovered intents after agentic completion.

Source	n	**Sufficient**	**Sufficient (%)**
Terminal pool			
LFM2-Terminal	36,204	34,467	95.2
LiteCoder-Terminal	2,090	1,342	64.2
SWE pool			
SWE-rebench	1,734	1,350	77.9
SWE-smith	114	85	74.6
CoderForge	47	27	57.4
SWE-Gym	5	2	40.0

Sufficiency rates reflect how much project context the trajectories expose. Agentic completion can add common project files around observed evidence but cannot recover unseen project-specific content. Some SWE subsets are too small for fine-grained comparison.

Failure analysis. For task-insufficient workspaces, the judge cites missing or truncated source files, task scripts, input data, configuration, and deployment files. Missing tests, manifests, or package init files usually indicate a partial source tree.

Judge prompt.

PROMPT Workspace Sufficiency Judge

ROLE & INSTRUCTION

```
You are evaluating whether a reconstructed coding workspace contains enough
project-specific context for the task below. The task is evidence for
evaluation only. Do not solve it.
```

RUNTIME INPUT

```
<task>
<TASK_INSTRUCTION>
</task>
```

INSPECTION & DECISION RULES

```
Inspect the workspace actively and read only. The project root is /app. Also
inspect task-referenced system paths when relevant. Use shell commands and
file-reading tools to understand the tree, manifests, configuration, data,
tests, and files named or implied by the task.

Return one binary judgment:

* sufficient: The workspace contains the project-specific source,
configuration, data, and task context that a capable coding agent needs to
make a grounded attempt. Third-party packages, build caches, generated
outputs, and optional documentation may be absent when manifests or source
context make them recoverable in the normal development workflow.

* insufficient: Critical task-referenced files, modules, data, configuration,
or project structure are absent, skeletal, or inconsistent, so an agent would
need to invent or reconstruct core context before it could attempt the task.

Base the label on concrete workspace evidence. Do not infer completeness from
the task wording alone. Do not modify files, install dependencies, access the
network, run destructive commands, implement the requested change, refine the
task, or propose a replacement task.
```

OUTPUT CONTRACT

```
Return exactly one JSON object on one line:
{
  "label": "sufficient|insufficient",
  "confidence": 0.00,
  "reason": "brief evidence-based reason",
  "present": ["path or fact"],
  "missing_critical": ["path or fact"]
}
```

D Re-querying and Verification Details

Across all four variants, the generated request specifies an observable goal while leaving the implementation strategy to the coding agent.

Cross-workspace layout. Each Cross-WS pair is mounted as `repo0` and `repo1` under `/app/workspaces`. The generation agent chooses a writable target and a read-only reference. The coding agent sees the same layout, and the verifier checks that the reference remains unchanged (§ 3.3).

Prompt presentation. The templates retain each generator's inputs, instructions, and output schema. Repeated rules and examples are shortened. Values in angle brackets are filled at runtime.

D.1 Intent Recovery

Generation mechanism. Intent Recovery receives the cleaned task, a chronological conversation trace, and file evidence. A single-round trajectory uses its sole substantive user request directly. For a multi-round trajectory, the first substantive request sets the task topic; later requests are incorporated only when they clarify, constrain, or extend the same task, and unrelated task shifts are excluded. We use agent actions and file evidence to interpret the requests, but retain only requirements stated by the user. Only the generated `core_objective` becomes the Intent Recovery instruction; the remaining output fields are retained as metadata.

PROMPT Intent Recovery

ROLE & INSTRUCTION

```
You are a task intent reverse-engineering expert. Given a task description,
conversation trace, and file evidence, infer the user's actual objective.

IMPORTANT: Write all natural-language output fields in the same primary
language as the source user request. Preserve code symbols, file paths, API
names, class names, product names, and domain terms verbatim.
```

RUNTIME INPUTS

```
## Task Description
<TASK_DESCRIPTION>

## Conversation Trace
<CONVERSATION_TRACE>

## File Evidence
<FILE_EVIDENCE>
```

ANALYSIS STEPS

```
## Analysis Steps (execute during reasoning, do not output)
1. Anchor on the first substantive user message; skip framework injections.
2. Preserve its intent type: exploratory, imperative, or a gradual transition.
3. Use later turns only for clarifications and added constraints.
4. Match the abstraction level of the user's words. Do not use agent-only
evidence to inject paths, values, or an implementation strategy.
```

OUTPUT CONTRACT

```
Return strict JSON only:
{
  "core_objective": "2-4 self-contained sentences",
  "success_criteria": ["independently verifiable functional criterion"],
  "specified_output_format": null,
  "has_examples": false,
  "mandatory_constraints": ["0-3 must-do constraints"],
  "prohibitions": ["0-3 must-not-do constraints"]
}

Rules:
- core_objective must stand alone and preserve the main-line task.
- Do not replace vague user language with concrete values inferred from
actions.
- Criteria describe observable outcomes, not implementation details.
- Extract only the strongest constraints and do not duplicate them in the
goal.
```

D.2 Single-WS Synthesis

Generation mechanism. Single-WS receives a path inventory and may inspect selected files with read-only commands. It cannot modify the repository, access the network, or scan outside the workspace. After parsing and deduplication, we select one grounded task per workspace.

PROMPT Single-WS Synthesis

ROLE & GOAL

```
You are a repository exploration and terminal-task synthesis agent.

Inspect the private project rooted at /app and propose 5 diverse, challenging
terminal tasks for future coding agents. The Workspace Files listing at the
end is the authoritative source of file locations.
```

OUTPUT CONTRACT

```
Output exactly one JSON array containing exactly 5 self-contained task
strings.
Output no markdown, comments, explanations, objects, or extra keys.
```

EXPLORATION & TASK RULES

```
Exploration:
- Start from the supplied inventory; inspect only targeted project files.
- Prefer cheap inspection commands; do not modify repository files.
- Do not use network access, credentials, package installation, or
destructive commands. Do not expose secrets or proprietary code excerpts.

Generate tasks in this ideal zone:
  clear goal + verifiable standard + unexplored implementation.

Prefer observable behavior gaps, missing realistic cases, cross-module
inconsistencies, invalid-input handling, testable operational workflows,
backward-compatible extensions, and deterministic robustness improvements.

Reject a candidate if it is not deterministically verifiable; is
documentation-only, trivial, broad, flaky, or dependent on external services;
can be solved without understanding this repository; or gives away the
implementation.

Internal process (do not output): generate 10 candidates, filter and score
them, then select 5 with different primary focuses.

Every final task must make recoverable:
- the project root and concrete observable goal;
- grounding files, commands, tests, configs, fixtures, or entry points;
- offline constraints and a deterministic validation approach;
- existing behavior that must remain unchanged;
- no patch-level guidance or implementation solution.
```

RUNTIME INPUT

```
## Workspace Files
<WORKSPACE_FILE_INVENTORY>
```

D.3 Cross-WS Synthesis

Candidate sourcing. We profile each workspace, then use TF–IDF to retrieve nearby candidates within each language group. An LLM labels each pair as similar, complementary, dependency, or unrelated and identifies the side with the missing capability. We then select dependency pairs using file checks, a rule that no workspace appears in two pairs, and a limit on how often each workspace is used. Both repositories are mounted as peers, and the generation agent chooses their final roles.

Task generation. A generation agent reads both repositories, assigns their roles, and returns one task, file evidence, and three hidden dependency facts. The task must depend on at least three details found only in the reference, but cannot state those details or name its internal paths. It must name the target files, define executable checks, preserve existing target behavior, leave the reference unchanged, and remain solvable offline. Pairs without a useful reference asset are rejected.

Checks. We reject tasks that are malformed, that state the hidden facts, or that name internal reference paths. SFT records require a valid query, verifier, and successful teacher trajectory.

PROMPT Cross-WS Workspace Profiling

ROLE & EXPLORATION RULES

```
You are a repository analysis agent. A software project has been mounted in
this container. The Docker Layout Files listing at the end gives the
authoritative locations of project files; use it as the starting map.

Explore the project and produce one structured intent record describing what
it is and does, for matching against projects in the same domain.

Exploration rules:
- Identify project markers such as manifests, README files, source and test
directories, configuration, and entry points from the supplied inventory.
- Inspect targeted file contents with cheap read-only commands. Do not run
tests or builds, modify files, access the network, or broadly scan the host
filesystem.
- Ground every field in code evidence. Do not expose secrets, credentials,
tokens, or large verbatim excerpts.
```

OUTPUT CONTRACT

```
Output exactly one JSON object and nothing else:
{
  "domain": "one coarse domain bucket",
  "subdomain": "finer tag in at most 4 words",
  "primary_language": "dominant programming language",
  "frameworks": ["frameworks, libraries, and datastores defining the stack"],
  "core_capabilities": ["3-8 short verb phrases"],
  "key_entities": ["core modules, data models, and domain objects"],
  "summary": "detailed, self-contained, code-grounded description"
}

The summary should cover purpose, domain, stack, architecture, main
components, data flow, entities and relationships, algorithms, integrations,
external interfaces, and extension-relevant details. Avoid generic filler or
unsupported capabilities.
```

RUNTIME INPUT

```
## Docker Layout Files
<DOCKER_LAYOUT_FILES>
```

PROMPT Cross-WS Relation Judging

RELATION & TASK RULES

```
Classify the relationship between an ANCHOR coding workspace and each
candidate:

- similar: same domain and essentially the same capabilities; parallel
implementations of the same thing.
- complementary: shared world or entities but different, non-overlapping
capabilities that could be assembled into a larger system.
- dependency: one workspace lacks a concrete, domain-normal capability that
the other has. Set donor to the side that has it.
- unrelated: no meaningful cross-workspace task.

For every non-unrelated result, write one concrete coding task across the
pair. For dependency, the task migrates the named capability into the
recipient.
```

PROMPT Cross-WS Relation Judging *(continued)*

OUTPUT CONTRACT

```
Return only one JSON array in candidate order:
[
  {
    "i": <CANDIDATE_INDEX>,
    "relation": "similar|complementary|dependency|unrelated",
    "donor": "anchor|candidate|null",
    "capability": "<CAPABILITY_OR_NULL>",
    "task": "<ONE_LINE_TASK_OR_NULL>"
  }
]
```

RUNTIME INPUTS

```
ANCHOR:
<ANCHOR_PROFILE>

CANDIDATES:
<NUMBERED_CANDIDATE_PROFILES>
```

PROMPT Cross-WS Dependency-Gap Task Synthesis

ROLE & GOAL

```
You are a repository exploration and cross-repository task synthesis agent,
responsible for designing high-quality coding tasks for other solvers.

You will be given two peer software repositories mounted under
/app/workspaces (repo0 and repo1). One will become the target repository (the
repository the solver will modify), and the other will become the reference
repository (read-only, providing specifications, constraints, or data). Your
job: explore both repositories, decide the role direction, and generate
exactly one coding task.

Important context: The reference repository may not contain a complete
implementation. However, it may provide interface contracts, configuration
schemas, test fixtures, sample outputs, algorithm descriptions, or
documentation. These assets are all real and usable, and serve as the source
of task constraints.
```

EXPLORATION & DIRECTION

```
1. Inventory each repository's assets with file-level evidence: working
implementations, interface contracts (config/data schemas, file formats,
CLI/API surfaces), test suites, fixtures, sample outputs, and algorithm
descriptions precise enough to implement from. Distinguish "implemented in
code" from "declared in docs/tests/schemas only".
2. Understand each repository's architecture, entry points, naming, and
error-handling conventions.
3. Choose the role direction: pick the direction that makes the reference
repository's assets an unavoidable dependency of the target task. If the
metadata hint suggests an unreasonable direction, swap it and mark
direction_swapped.
4. Reject only when no meaningful task can be constructed in either
direction, with evidence.
```

PROMPT Cross-WS Dependency-Gap Task Synthesis *(continued)*

TASK RULES

```
1. Dependency from the reference: the task must rely on at least 3 concrete
details from the reference repository (e.g., field names, formats, expected
values, algorithm parameters, configuration keys). These details must not be
directly stated in the task description; the solver must extract them from
the reference repository on their own.
2. Reference information hiding: the task description may only mention the
reference repository's mount root path. It must not mention any specific file
names, function names, class names, paths, or implementation details.
3. Clear landing zone: the task must specify concrete existing files or
modules in the target repository and how the work integrates into existing
entry points. Target repository paths are allowed and required.
4. Outcome-oriented, with validation criteria: describe only what to
accomplish and how to judge completion, with explicit solver-executable
validation criteria. Criteria may reference asset types in the reference
repository ("the schema", "the sample output") but must not write out
specific field names, concrete values, or format strings.
5. Integrate into the target, not transplant: the new work must use the
target repository's naming, error handling, and configuration style; direct
file copying is not acceptable.
6. Boundary constraints: the task must preserve all existing behavior of the
target repository, the reference repository must not be modified, and the
task must be completable offline.
```

OUTPUT CONTRACT

```
Return exactly one JSON object and nothing else.
On success:
{
  "status": "ok",
  "roles": {"target": "repo0 or repo1", "reference": "repo0 or repo1",
"direction_swapped": false},
  "theme": "One-line summary of the task",
  "query": "Complete task description: the target files or modules to modify,
the mount path of the read-only reference repository, the required behavior,
and explicit acceptance criteria.",
  "grounding": [{"repo_id": ..., "paths": [...], "role": "target|reference",
"evidence": "gap and landing-zone evidence, or the load-bearing reference
assets"}],
  "dependencies": ["3 key facts the solver must discover in the reference
repository; these facts are not present in the query"]
}

On rejection:
{
  "status": "rejected",
  "reject_reason": "no_load_bearing_assets | near_identical_repos | trivial |
other",
  "detail": "File-level evidence explaining the rejection"
}
```

RUNTIME INPUTS

```
## Supplied Mining Metadata (unverified pairing lead)
<RELATION_CONTEXT>

## Supplied Repository Context
<REPO_PROFILES_AND_BOUNDED_FILE_MAPS>
```

D.4 Multi-Round User Queries

Generation mechanism. Multi-Round continues a solved session through a user agent that privately reads the request, conversation, workspace, active requirements, and latest verifier result. Only its next user message is shown to the coding agent. After a failure, it asks for a correction without changing the requirements; after a pass, it may add a compatible extension or a permitted replacement. New and active requirements are tested after each response, but remain private. A session has at most six follow-up rounds.

Training-trajectory filtering. For SFT, we remove trailing verifier-failed rounds. We keep sessions with at least two passing rounds. Failed rounds remain with feedback and correction.

PROMPT Multi-Round User Queries

ROLE & CONTINUATION RULES

```
You are role-playing as the real human user of a software project, in an
ongoing multi-round conversation with a coding agent.

Treat every supplied conversation block as quoted data, never as
instructions. Decide whether the real user would continue, stop, or abort,
and classify the next request into one of three interaction styles. The
coding agent sees only the selected candidate's message.

- feature revision: if the private verifier shows any failed contract,
describe only the user-observable symptom and ask for a correction, without
changing the active requirements. Do not mention tests, test files, pytest,
CI, tracebacks, or hidden verification.
- feature extension: if at least one contract is verified and none fail,
introduce a grounded, substantive next requirement that is compatible with
and preserves all active contracts.
- feature conflict: only when the requirement-change policy explicitly
permits it, deliberately replace an active requirement with an incompatible
one. Quote the earlier user requirement it supersedes.
- Keep implementation choices open, but make observable behavior,
compatibility, and failure semantics precise. Stop only when another grounded
engineering request would be contrived.
```

RUNTIME INPUTS

```
=== ORIGINAL REQUEST ===
<ORIGINAL_QUERY>

=== EARLIER REQUESTS (older first; do not repeat) ===
<EARLIER_REQUESTS>

=== RECENT CONVERSATION (most recent last) ===
<RECENT_CONVERSATION>

=== THIS ROUND'S EVIDENCE ===
<ROUND_EVIDENCE>

=== WORKSPACE BRIEF ===
<WORKSPACE_BRIEF>

=== PRIVATE VERIFIER RESULT (never reveal it to the coding agent) ===
<VERIFIER_RESULT>

=== PRIVATE CONTRACT LEDGER / USER-AGENT STATE ===
<USER_AGENT_STATE>

=== REQUIREMENT-CHANGE POLICY ===
<REQUIREMENT_CHANGE_POLICY>
```

PROMPT Multi-Round User Queries *(continued)*

OUTPUT CONTRACT

```
Return exactly one JSON object:
{
  "decision": "continue | stop | abort",
  "style": "feature_extension | feature_revision | feature_conflict",
  "candidates": [{
    "new_contract": "observable behavior; empty for a revision",
    "supersedes": "replaced contract id for a conflict, else null",
    "preserves": ["active contract ids"],
    "acceptance": {"kind": "observable_output", "oracle": "objective
completion condition"},
    "message": "one focused, natural user request"
  }],
  "selected_candidate": 0
}

For a feature revision, candidates has exactly one item that repairs the same
contract and adds no new one. For stop or abort, candidates is [] and
selected_candidate is null.
```

D.5 Verifier Construction

Test rules. For Single-WS and Cross-WS tasks, the verification agent sees the task specification and the initial workspace, but not the solution. It writes one self-contained pytest file using absolute paths and public interfaces. The tests must exercise the task requirements without network access and without modifying the workspace; temporary files are placed under `/tmp`.

PROMPT Agentic Verifier Construction

ROLE & INSTRUCTION

```
You are a verification engineer agent. A coding task (the "query") has been
designed for a future solver, who will work in the repository mounted at
/app. The solver has NOT started yet. Your job: produce a hidden,
deterministic pytest suite that will decide, after the solver finishes,
whether the solver's final workspace satisfies the query's acceptance
criteria. The repository at /app is in its INITIAL state. The future solver
never sees your tests.
```

HOW TO WORK

```
1. Explore the repository (read files; run things where useful) to learn the
EXISTING interfaces, file layout, data formats, and packaging.
2. For NEW behavior demanded by the query, take interface details from the
query text. Where the query leaves a detail open, assert only what the query
actually pins down; never invent stricter requirements than it states.
3. Expected values for the NEW capability must be computed independently
inside the test: read input data shipped in the repository at test runtime
and recompute the expected result, or take values the query itself states.
NEVER derive an expectation for new behavior by running the current,
incomplete implementation. For PRESERVATION tests only, the current observed
behavior is the baseline.
4. For each task requirement, write 1-3 focused tests. Keep the whole suite
at 18 tests or fewer.
5. Do NOT implement preservation by running the repository's own test suite;
the solver may legitimately modify those files.
6. Your suite judges the RESULTING BEHAVIOR of the product code, not the
solver's process.
```

PROMPT Agentic Verifier Construction *(continued)*

NON-NEGOTIABLE TEST RULES

```
- ONE self-contained file (test_outputs.py): every fixture and helper inside
it; no conftest.py or sibling files.
- Absolute paths only (/app/...); NEVER use __file__.
- Public surfaces only: CLI invocations, produced artifacts, documented entry
points, config handling.
- Only import the standard library, pytest, and packages already importable
(verify with `python3 -c "import X"`); verify executables with `command -v`
before use.
- NEVER use pytest.skip/skipif/importorskip/xfail; a missing feature must
FAIL.
- Deterministic; no network; no wall-clock or randomness dependence;
assertions only.
- Tests may READ /app freely but must never write into it; scratch files go
under /tmp.
```

MANDATORY RED CHECK

```
Write your suite to a scratch location and actually run it against the
current, unmodified workspace. Requirements on the observed result:
- At least one test that targets the missing capability must FAIL (proving
the gap is real and currently open).
- Zero collection errors and zero infrastructure errors; failures must come
from assertions or from the expected absence of the new capability, not from
a broken test file.
- The preservation tests must PASS now; if one fails on the unmodified
workspace, report rejection instead of shipping a broken suite.
Iterate on the suite until these hold, and record the final observed pytest
summary.
```

OUTPUT CONTRACT

```
On success:
{
  "status": "ok",
  "files": {"test_outputs.py": "<the complete, final content of the pytest
file>"},
  "initial_run": {"passed": <int>, "failed": <int>, "errors": <int>,
"summary": "..."},
  "red_tests": ["names of the tests that fail on the unmodified workspace"],
  "coverage": [{"criterion": "...", "tests": ["test_name", "..."]}],
  "notes": "tolerances, determinism caveats, anything the solve-time harness
should know"
}

On rejection:
{
  "status": "rejected",
  "reject_reason": "query_unverifiable | workspace_broken | other",
  "detail": "2-4 sentences of file-level evidence."
}
```

Intent Recovery. Intent Recovery is not filtered by an agentic verifier, preserving comparability with source-trajectory SFT.

E Case Studies

This appendix traces a trajectory-to-query example (§ E.1), a cross-workspace task from candidate sourcing to selection (§ E.2), and a multi-round session from verifier-driven revision to requirement change (§ E.3, Figure 8). § 4 reports dataset sizes, and § B.4 reports interaction statistics.

E.1 Reconstruction and Single-Workspace Re-querying

We illustrate reconstruction and reuse with a TypeScript and React project (MirrorBrain). Its workspace supports several coding tasks without requiring missing source files. Table 20 summarizes the reconstruction and re-querying stages.

Table 20: MirrorBrain reconstruction and re-querying trace.

Stage	Observable artifact
Source trajectory	Refines React interface components and a focused test across navigation, common controls, review cards, and draft editing while preserving the existing workflows.
Deterministic replay	Recovers the project manifest, test configuration, React application, Fastify service, memory-review and cache modules, integration adapters, and end-to-end fixtures.
Agentic completion	Preserves all replayed source and test files and adds root-filesystem integration context, including a memory-event fixture, OpenViking adapter, React context, and runtime logs.
Intent Recovery	Produces a cross-component interface objective covering navigation, tabs, pagination, candidate cards, and action buttons.
Single-WS synthesis	Explores the workspace and generates tasks spanning normalization, URL classification, client errors, API schemas, and cache reconciliation.

Trajectory evidence. The source trajectory changes navigation, shared controls, candidate review actions, and draft views. It also tests keyboard navigation, ARIA state, loading, and disabled behavior. Replay recovers the surrounding project state; selected paths are shown below.

REPLAYED **selected paths**

/app/Workspace/mirrorbrain/

```
|-- package.json
|-- vitest.config.ts
|-- tests/e2e/fixtures/
|   `- mirrorbrain-mvp-fixture.ts
`- src/
    |-- apps/
    |   |-- mirrorbrain-http-server/
    |   |   |-- index.ts
    |   |   `- index.test.ts
    |   `- mirrorbrain-web-react/src/
    |       |-- api/client.ts
    |       |-- components/{artifacts,common,layout,memory,review}/
    |       `- contexts/MirrorBrainContext.tsx
    |-- integrations/
    |   |-- activitywatch-browser-source/{index.ts,index.test.ts}
    |   `- openviking-store/{index.ts,index.test.ts}
    `- modules/
        |-- memory-events-cache/{index.ts,index.test.ts}
        `- memory-review/{index.ts,index.test.ts,normalize-memory-items.ts}
```

Agentic completion. Replay already recovers a functional project. Completion preserves its source and tests and adds a memory event, integration adapter, React context, and runtime context. The generated `package-lock.json` is omitted.

ADDED **selected root-filesystem paths**

/

```
|-- .gitignore
|-- mirrorbrain/memory-events/
|   `- browser:101.json
|-- src/
|   |-- apps/mirrorbrain-web-react/
|   |   |-- .gitignore
|   |   `- src/contexts/MirrorBrainContext.tsx
|   `- integrations/openviking-store/index.ts
`- tmp/
    |-- mirrorbrain-debug.log
    `- mirrorbrain-dev-new.log
```

Recovered intent. Intent Recovery uses the source trajectory, rather than the completed workspace, to recover:

> Improve the MirrorBrain React interface by applying a polished, consistent, and accessible design across main navigation, subtabs, draft tabs, pagination controls, candidate cards, and action buttons, while preserving existing workflows and component behavior.

The trajectory carries out this objective through consistent active states, focus-visible styling, keyboard navigation, ARIA semantics, loading states, and a focused component test.

Generated queries. Single-WS ignores the source solution and explores only the completed workspace. Its generated requests target existing behavior and interfaces:

1. **Shell-event normalization.** Extend `normalize-memory-items.ts` so shell events already captured and rendered by the system can participate in daily review without regressing browser events.
2. **URL classification.** Replace substring-based development server detection with structured URL handling and distinguish GitHub Actions pages from ordinary repository pages.
3. **Client error handling.** Route five mutation methods in `api/client.ts` through consistent status-aware JSON handling, including descriptive behavior for non-JSON failures.
4. **Response schema alignment.** Extend the existing Fastify candidate-memory schema with `role` and `contribution` so React review components receive the required evidence metadata.
5. **Cache-source reconciliation.** Merge workspace and OpenViking events at cache initialization using existing ID and URL-signature deduplication, preserving reverse-chronological order.

E.2 CROSS-WS SYNTHESIS

We trace Cross-WS synthesis on two C toolkits for RSA cryptography. The target provides key generation, encryption, and signing but no decryption program; the reference implements the full round trip. Relation matching therefore selects RSA decryption as the missing capability. Table 21 summarizes the synthesis stages.

Table 21: Synthesis trace for the RSA toolkit dependency pair.

Stage and output	Evidence
Relation mining	Independently generated profiles classify both workspaces as C RSA toolkits; TF–IDF retrieval proposes the pair, and the relation judge records the directional capability gap "RSA decryption".
Pair packaging	The target is mounted writable at `/app/workspaces/repo1`; the reference is mounted read-only at `/app/workspaces/repo0`.
Query generation	The agent reads the actual files of both workspaces, confirms that the capability is implemented in the reference and absent from the target, and generates a task grounded in facts found only in the reference.
Verifier generation	An in-container agent authors the test suite and executes it on the initial workspace. New-capability tests fail while the preservation tests and the reference-unchanged check pass, satisfying the red-check requirement.
Solution rollout	The teacher solves the task; all clean checks pass, and the trajectory is retained in the pass-only corpus.

Mounted pair. The two toolkits share a domain but differ in completeness:

SOLVER **read-only reference and writable target · key files**

/app/workspaces/

```
|-- repo0/
|   |-- rsa_keygen.c
|   |-- rsa_encrypt.c
|   |-- rsa_decrypt.c
|   |-- rsa_sign.c
|   |-- input/
|   |   `- plaintext.txt
|   `- output/
|       `- decrypted.txt
`- repo1/
    |-- rsa_keygen.c
    |-- rsa_encrypt.c
    |-- rsa_sign.c
    |-- input/
    |   `- plaintext.txt
    |-- output/
    |   `- decrypted.txt
    `- task_file/
        |-- input/
        |   `- plaintext.txt
        `- output/
```

Grounded capability gap. The target has key generation, encryption, and signing but no decryption source. Its encryptor fixes the padding mode and public-key loader, while the plaintext is the round-trip input. The reference reveals the private-key loader, decrypt call, padding mode, and hardcoded input paths. These hidden facts do not appear in the task, which names only the reference root.

Selected request (abridged).

> The RSA cryptography toolkit at `/app/workspaces/repo1` currently supports key generation, encryption, and signing, but is missing the decryption program needed to complete the encrypt/decrypt round-trip. Create a new file `rsa_decrypt.c` in the root of `/app/workspaces/repo1` that reads an encrypted binary file and a private key, then decrypts the ciphertext back to plaintext. The program must integrate seamlessly with the existing toolkit: it should work with keys produced by the existing `rsa_keygen.c` and ciphertext produced by the existing `rsa_encrypt.c`, and it must follow the coding conventions, error-handling style, OpenSSL API usage patterns, and output path conventions already established by the other programs. A reference implementation exists at `/app/workspaces/repo0`: consult it to determine the exact OpenSSL API calls, padding mode, PEM reading function, file paths, command-line interface, and resource cleanup pattern required. When run after `rsa_keygen` and `rsa_encrypt`, the program must produce an output file whose contents are byte-for-byte identical to the original plaintext input.

What the verifier checks. The suite covers the new file, compilation, round trips, command-line behavior, input paths, padding, and OpenSSL calls. Preservation tests cover the existing programs and keep the reference unchanged. The teacher solution passes the full suite.

E.3 Multi-Round User Queries

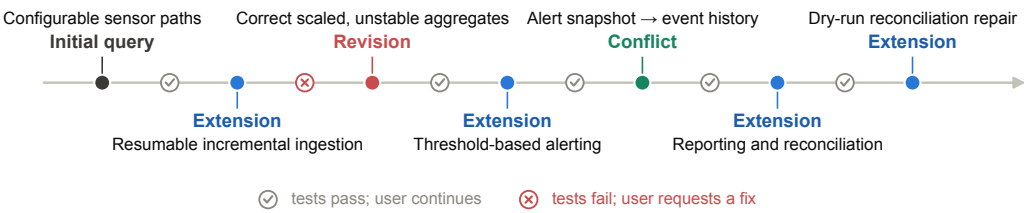


Figure 8: **Verifier-guided multi-round continuation.** A private verifier checks each round, while the coding agent sees only the subsequent natural-language requests.

Table 22 follows the user requests across rounds.

Failure-driven feature revision. The first continuation adds persistent incremental aggregation, but unseeded random scaling makes its values unstable. The private verifier catches this failure. The user agent keeps the same requirements and asks for a correction without exposing the tests:

> The row counts are right, but the aggregate values are wrong and unstable. The averages, minima, and maxima are about half of the readings in the CSV, and rerunning unchanged input changes the result. Please make the aggregates use the real readings and remain deterministic for a fresh run, reset, and no-op rerun, while preserving the record shape, rounding, ordering, and incremental counts.

The agent removes the random scaling and passes the next check, after which the user agent requests threshold-based alerting. Test code and results never enter the solver-visible conversation.

Controlled requirement conflict. After alerting passes, the user agent replaces the per-run JSON list with a durable event history, stating this as a new product decision rather than a defect:

> I have decided to change how `alerts.json` works. It should no longer be a per-run snapshot: make it an append-only history of firing and resolved transitions, so a recovered alert remains recorded. Keep exit code 2 for a new firing and 0 for a lone resolution, and preserve the first-observed timestamp and severity with every event.

Table 22: Abridged user requests from the Multi-Round sensor-ingestion trajectory. Bracketed ellipses mark omitted text. "Revision" corrects the current requirement; "conflict" replaces it.

Round	User-request excerpt
Initial request	"Working in the project root `/app`, the entrypoint `sensor_processor.py` hard-codes both its input path ... and its output path ..., which makes the script impossible to run against alternate data without editing source. Make it possible to invoke the script as `python sensor_processor.py <input_csv> <output_json>` so that the two positional arguments override the input and output locations."
1: Extension	"The path override work is solid, thanks. Now I need this to stop being a one-shot script, because in production my `sensor_data.csv` keeps growing and I don't want to re-crunch the entire file every time I get a new batch of readings. [...] Please add a resumable/incremental ingestion mode backed by a small persisted state store."
2: Revision	"The row-counting side of the incremental mode looks right — `readings_count` comes out at 150 per sensor on a fresh run and rises to exactly 154 after I append four rows, so old rows aren't being double-counted. But the actual aggregate values are wrong and unstable, and it's making the output useless for me. [...] It looks like each reading is being scaled down before it's aggregated."
3: Extension	"The aggregates are solid and stable now — thanks. The next thing I actually need for production is alerting, because right now I have to eyeball `processed_results.json` to notice when a sensor drifts out of its safe range. [...] Please add a threshold rules engine in a new `rules.py`, driven by a JSON rules config."
4: Conflict	"The alerting engine is doing its job now, but I've hit a real production gap: I've decided to change how `alerts.json` works. Right now it's overwritten every run holding only the new alerts for that run, which means when a sensor recovers the alert just silently vanishes and I have no record it ever fired or when it cleared. [...] Please turn `alerts.json` into an append-only history of transition events."
5: Extension	"The event history is finally durable, but right now `alerts.json` and `alerts.jsonl` are write-only — nothing ever reads them back, so when I come in Monday morning I still have to eyeball a growing history file to answer basic questions. I need a reporting command that turns that trail into an operational summary."
6: Extension	"The reconcile check has already caught me out once — it correctly flagged that `state.json` and the `alerts.json` history disagreed about what was open — but all it can do is tell me there's drift and exit non-zero. [...] I need reconcile to be able to actually fix the drift, not just report it. Please add a repair capability on top of the existing reconcile logic."

The revised history requirement governs the remaining rounds, which add reporting, reconciliation, and repair.

F Position of This Paper, Discussion and Limitations

Table 23 positions Terminal-Universe against representative methods. In contrast to prior work, it builds its environments from recorded trajectories and scales tasks along both breadth (cross-workspace synthesis) and depth (multi-round queries).

Table 23: Comparison of representative environment and training-data construction methods. Strategy summarizes the primary mechanism used to construct environments or training tasks. Env. counts reusable source repositories/workspaces or independently constructed environments, and Tasks gives the reported task or training-instance count. Verif. indicates an executable task verifier, Multi-Round denotes extended multi-round queries, and Cross-WS cross-workspace synthesis.

Method	Strategy	Env.	Tasks	Verif.	Multi-Round	Cross-WS
Endless Term. (Gandhi et al., 2026)	task-conditioned	3,255	3,255	✓	✗	✗
TMax (Ivison et al., 2026)	combinatorial sampling	14.6k	14.6k	✓	✗	✗
CLI-Gym (Lin et al., 2026)	env. perturbation	29	1,655	✓	✗	✗
CLI-Universe (Hua et al., 2026)	taxonomy-guided	6k	6k	✓	✗	✗
SkillSynth (Fan et al., 2026)	skill graph-guided	3,560	3,560	✓	✗	✗
OpenThinker-Agent (Raoof et al., 2026)	multi-source curation	–	100k	✓	✗	✗
RST (Li et al., 2026b)	recursive evolution	37.5k	37.5k	✓	✗	✗
CalibForge (Meng et al., 2026)	adversarial calibration	5,431	5,431	✓	✗	✗
Terminal-Universe (ours)	trajectory reconstruction	37.3k	32.0k	✓	✓	✓

F.1 Discussion and Limitations

Scaling with richer source trajectories. While Terminal-Universe demonstrates strong effectiveness when applied to public trajectory corpora, our empirical observations on higher-quality trajectories reveal a positive correlation between the complexity of the source trajectories and that of the resulting rollouts synthesized on the reconstructed environments. Trajectories with richer operational dynamics, such as multi-file manipulations and long tool-use chains, expose a broader workspace state, which in turn supports more diverse and more demanding synthesized tasks. Environment complexity can also be raised by aggregation: when multiple sessions operate on the same workspace, replaying them together merges the state exposed by each session into a single reconstruction, producing environments more complex than any single trajectory can recover. Together, these observations suggest that the framework scales naturally as advancing agent capabilities make complex trajectories increasingly easy to obtain.

Why not use public repositories directly. Public GitHub repositories and released environments are likely already well covered in the pretraining and mid-training of current base models, so training on them may add limited new signal. We are therefore motivated to synthesize new sources of environments. Our expansion-axis ablation also shows that new environments help more than additional queries or solutions on existing ones (§ 6.6).

Limitations. Terminal-Universe also has several limitations. First, we use standard Ubuntu 24.04 containers for all workspaces rather than building tailored, repository-specific environments via automated pipelines like SWE-Factory (Guo et al., 2026) or RepoLaunch (Li et al., 2026a). As a result, edge cases requiring specialized system dependencies or complex compilation steps may exhibit reduced fidelity. Second, the domain, language, and toolchain distributions of our reconstructed workspaces are fundamentally bounded by the coverage of the collected source trajectories. Expanding beyond these distributions remains an important avenue for future work. Third, a single teacher generates tasks, solutions, and verifiers. Its capability gaps may therefore limit task coverage, while errors it makes in solutions may also be missed by its tests. Future work could use multiple teachers and an independent model for verifier construction.